On the Verification of Memory Management Mechanisms

Iakov Dalinger* Mark Hillebrand* Wolfgang Paul

January 2005

Id: verificationmm.tex,v 1.35 2005/01/27 14:11:10 mah Exp

Abstract

We define physical machines as processors with physical memory and swap memory; in user mode physical machines support address translation. We report about the formal verification of a complex processor supporting address translation by means of a memory management unit (MMU). We give a paper and pencil proof that physical machines together with appropriate page fault handlers simulate virtual machines.

Contents

*Work partially funded by the German Federal Ministry of Education, Science, Research and Technology (BMBF) in the framework of the Verisoft project under grant 01 IS C38. Work of the second author was also partially funded by IBM Entwicklung GmbH Boeblingen. The responsibility for this article lies with the authors.

1 Introduction

1.1 The challenge of verifying entire systems

In the spirit of the famous CLI stack [Boy89] the research of this paper aims at the formal verification of entire computer systems consisting of hardware, compiler, operating system, communication system, and applications. Working with the Boyer-Moore theorem prover [BM88] the researchers of the CLI stack project succeeded as early as 1989 to prove formally the correctness of a system which contained among others the following components: (i) a non pipelined processor [Hun89], (ii) an assembler [Moo89], (iii) a compiler for an imperative language [You89] with the only data type int, assignments, if then else, while loops, and function calls, (iv) a rudimentary operating system kernel [Bev89] written in machine language. This kernel provides scheduling for a fixed number of processes; each process has the right to access a fixed interval of addresses in the processors physical memory. An attempt to access memory outside these bounds leads to an interrupt. Interprocess communication and system calls apparently were not provided.

In the years from 1989 to 2002 to the best of our knowledge no verification project aiming at the formal verification of entire computer systems was started anywhere. In [Moo03] the principal investigator of the CLI stack project J S. Moore declares the design and formal verification of practical embedded systems 'from the transistor level to the software' a grand challenge problem. A central goal of the Verisoft project [Ver03], funded by the federal German Government and started in 2003, is to solve this grand challenge problem.

This paper makes two necessary steps towards the verification of entire complex systems. (i) We report about the formal verification of a processor with memory management units (MMUs). MMUs provide hardware support for address translation; address translation is needed for the implementation of address spaces provided by modern operating systems. (ii) We present a paper and pencil correctness proof for a virtual memory

emulation based on a very simple page fault handler. As the formal treatment of I/O-devices is an open problem we have to state the correctness of a driver for the swap memory as an axiom.

In subsequent papers we will address the verification of a compiler for a C-like language with in-line assembler code and of an operating system kernel. For preliminary versions of these results see [**?**, **?**, **?**].

1.2 Overview of this paper

In Section 2 we briefly review the standard formal definition of the DLX instruction set architecture (ISA) for virtual machines. We put emphasis on the handling of internal and external interrupts. In Section 3 on physical machines we enrich the ISA by the standard mechanisms for operating system support: (i) user mode / system mode and (ii) address translation in user mode. In Section 4 we present a (non-optimized) construction of an MMU and prove its correctness under nontrivial operating conditions. Note that in pipelined processors separate MMUs are used for instruction fetch and load / store. In Section 5 we show how the operating conditions for both MMUs can be guaranteed by a combination of hardware mechanisms and software conventions. Section 6 gives the main new arguments of the processor correctness proof assuming that the software conventions are met. In Section 7 we present a simple page fault handler which obeys the software convention. We show that a physical machine with this page fault handler emulates a virtual machine. In Section 8 we present conclusions and further work.

1.3 Related work

The processor verification presented here extends work on the VAMP processor presented in [BJK$^+$03, Bey04]. The treatment of external interrupts is in the spirit of [SH98, MP00]. Formal proofs are in PVS [OSR92] and—except for some limited use of PVS's model checker—interactive. We stress that some central lemmas in [SH98, BJK$^+$03] (e.g. correctness of Tomasulo schedulers) have similar counterparts, which can be proven using the very rich set of automatic methods for hardware verification (e.g. [**?**]). How to profit from these automatic methods in correctness proofs of entire processors continues to be an amazingly difficult topic of research. Some recent progress is reported in [ACHK04].

As for the new results of this paper: we are not aware of previous work on the verification of MMUs. We are also not aware of previous theoretical work on the correctness of virtual memory emulations.

2 Virtual machines

2.1 Notation

We denote the concatenation of bit strings $a \in \{0,1\}^n$ and $b \in \{0,1\}^m$ by $a \circ b$. For bits $x \in \{0,1\}$ and nonnegative natural numbers $n \in \mathbb{N}^+$

we define inductively $x^1 = x$ and $x^n = x^{n-1} \circ x$. Thus, for instance $0^5 = 00000$ and $1^2 = 11$.

Overloading symbols like $+$, $\cdot$, and $<$ we will allow arithmetic on bit strings $a \in \{0,1\}^n$. In these cases arithmetic is binary modulo 2^n. We will consider $n = 32$ for addresses / register contents and $n = 20$ for page indices.

We model memories m as mappings from addresses a to values $m(a)$. For natural numbers d we denote by $m_d(a)$ the content of d consecutive memory cells starting at address a:

$$m_d(a) = m(a + d - 1) \circ \cdots \circ m(a)$$

Page size will be $4K = 2^{12}$. We split virtual addresses $va = va[31:0]$ into page index $va.px = va[31:12]$ and byte index $va.bx = va[11:0]$. Thus, $va = va.px \circ va.bx$.

2.2 Specifying the instruction set architecture

Virtual machines are the hardware model visible for user processes. Important parameters of such a machine are the following:

- The number V of pages of accessible virtual memory. This defines the set of accessible virtual addresses

$$VA = \{a \mid 0 \leq a < V \cdot 4K\}$$

- The number $e \in \mathbb{N}$ of external interrupt signals.
- The status register $SR \in \{0,1\}^{32}$. This is the vector of mask bits for the interrupts. In physical machines it comes from the status register.
- The set $VSA \subseteq \{0,1\}^5$ of addresses of user visible special purpose registers. Table 1 shows the entire set of special purpose registers, that will be visible for a physical machine. For the virtual machine only the registers RM, $IEEEf$, and FCC will be visible. Hence $VSA = \{00110, 00111, 01000\}$.

Formally, the configurations of a virtual machine is a 6-tuple

$$c_V = (c_V.PC, c_V.DPC, c_V.GPR, c_V.SPR, c_V.vm, c_V.p) \ .$$

The individual components are:

- the program counter $c_V.PC \in \{0,1\}^{32}$ and the delayed program counter $c_V.DPC \in \{0,1\}^{32}$. Both are used to implement the delayed branch mechanism (see Chapter 4 in [MP00] for details);
- The general purpose register file $c_V.GPR : \{0,1\}^5 \to \{0,1\}^{32}$. Registers are 32 bits long, register 0 always contains 0^{32}.
- The special purpose register file $c_V.SPR : VSA \to \{0,1\}^{32}$. We also refer to special purpose register by $c_V.x = c_V.SPR[x]$ where x is one of the synonyms from Table 1.
- The byte addressable virtual memory $c_V.vm : VA \to \{0,1\}^8$.

Address	Synonym	Meaning
00000	SR	Status register
00001	ESR	Exception status register
00010	ECA	Exception cause register
00011	EPC	Exception PC
00100	EDPC	Exception DPC
00101	Edata	Exception data
00110	RM	Rounding mode
00111	IEEEf	IEEE flags
01000	FCC	Floating point condition code
01001	pto	Page table origin
01010	ptl	Page table length
01011	EMODE	Exception mode
10000	MODE	Mode

Table 1: Special purpose registers. Indices 01100 to 01111 are not assigned.

- The write protection function $c_V.p : \{va.px \mid va \in VA\} \to \{0,1\}$. Virtual addresses on the same page have the same protection bit.

Let C_V be the set of virtual machine configurations. An instruction set architecture (ISA) is formally specified as a transition function $\delta_V : C_V \times \{0,1\}^e \to C_V$ mapping configurations $c_V \in C_V$ and a vector of external event signals $eev \in \{0,1\}^e$ to a next configuration $c'_V = \delta_V(c_V, eev)$. For the DLX instruction set we outline the formal definition of this function with an emphasis on the treatment of interrupts.

The instruction to be executed in configuration c_V is found in the four bytes in virtual memory starting at the address of the delayed PC

$$I(c_V) = c_V.vm_4(c_V.DPC)$$

The opcode consists of the leading six bits of the instruction

$$opc(c_V) = I(c_V)[31:26]$$

The type of an instruction determines, how the bits outside the opcode are interpreted. If the opcode consists of all zeros we have for instance an R-type instruction

$$\textit{R-type}(c_V) = (opc(c_V) = 000000)$$

Other instruction types are defined in a similar way. Many instructions can be decoded just from the opcode, e.g. a load word instruction is recognized by

$$lw(c_V) = (opc(c_V) = 100011)$$

For R-type instructions one has to refer to a secondary opcode. Depending on the instruction type the register destination address is found at different positions in the instruction

$$RD(c_V) = \begin{cases} I(c_V)[15:11] & \text{if } \textit{R-type}(c_V) \\ I(c_V)[20:16] & \text{otherwise} \end{cases}$$

In a similar way one can define register source addresses $RS1(c_V), RS2(c_V)$, the sign extended immediate constant $simm(c_V)$, etc. The effective address of a load or store instruction is computed as the sum of the general purpose register addressed by $RD(c_V)$ and the sign extended immediate constant

$$ea(c_V) = c_V.GPR(RD(c_V)) + simm(c_V)$$

A load word instruction stores four bytes of virtual memory starting at address $ea(c_V)$ into the general purpose register addressed by $RS1(c_V)$. This can be expressed by equations like

$$lw(c_V) \rightarrow (c'_V.GPR(RS1(c_V)) = c_V.vm_4(ea(c_V))) \ .$$

Components of the configuration that are not mentioned on the right-hand side of the implication are meant to be unchanged. This definition however ignores both internal and external interrupts; therefore even for virtual machines it is an oversimplification.

2.3 Interrupts

We define below a predicate $JISR(c_V, eev)$ (jump to interrupt service routine) depending on both the current configuration c_V and the current values $eev \in \{0,1\}^e$ of the external interrupt event signals. Only if this signal stays inactive does the above equation hold

$$(\neg JISR(c_V, eev) \wedge lw(c_V)) \rightarrow (c'_V.GPR(RS1(c_V)) = c_V.vm_4(ea(c_V)))$$

An activation of the $JISR$ signal has for physical machines a well defined effect on the program counters and on the special purpose registers. The effect on virtual machine computations however is that control is handed over to the operating system kernel. This effect can only be defined in a model, which includes the operating system kernel too.[1] For the same reason `rfe`-instructions (return from interrupt) are illegal for virtual machines.

For the definition of signal $JISR(c_V, eev)$ for physical machines, we consider the 32 interrupts from Table 2 with indices $j \in IP = \{0, \ldots, 31\}$. For virtual machines we ignore page fault interrupts, thus we only consider $j \in IV = IP \setminus \{3, 4\}$.

The activation of signal $JISR(c_V, eev)$ can be caused by the activation of external interrupt lines $eev[j]$ or by internal interrupt event signal $iev(c_V)[j]$. We define the cause vector by

$$ca(c_V, eev)[j] = \begin{cases} eev[0] & \text{if } j = 0 \ ; \\ eev[j-13] & \text{if } j \text{ external}, j \neq 0 \ ; \\ iev(c_V)[j] & \text{otherwise.} \end{cases}$$

Formally, external interrupts lines are external input signals for the next state computation. Internal interrupt event signals can be defined as

[1] See the lecture notes [SysArch04] (in German) for details.

Index j	Symbol	Meaning	Maskable	External
0	reset	Reset	No	Yes
1	ill	Illegal instruction	No	No
2	mal	Misaligned access	No	No
3	pff	Page fault during fetch	No	No
4	pfls	Page fault during load / store	No	No
5	trap	Trap	No	No
6	xovf	Fixed point overflow	Yes	No
7	fovf	floating point (FP) overflow	Yes	No
8	funf	FP underflow	Yes	No
9	finx	FP inexact result	Yes	No
10	fdbz	FP division by zero	Yes	No
11	finv	FP invalid operation	Yes	No
12	ufop	Unimplemented FP operation	No	No
13	timer	Timer	No	No
$j > 13$	$io[j]$	Interrupt from I/O device $j - 13$	Yes	Yes

Table 2: Interrupts

functions of the current configuration. A straightforward definition of the misalignment signal would for instance include terms like

$$\begin{aligned} mal(c_V) = iev(c_V)[2] &= (c_V.DPC[1:0] \neq 00) \vee \\ &\quad (lw(c_V) \wedge (ea(c_V)[1:0] \neq 00)) \vee \cdots \end{aligned}$$

For virtual machines, but not for physical machines, an attempt to read or write special purpose registers other than RM, $IEEEf$, FCC is illegal. Reading or writing these registers is achieved with commands `movi2s` or `movs2i`; special purpose registers are addressed with instruction field $SA(c_V) = I(c_V)[10:6]$. Thus a straightforward formal definition of the illegal instruction signal would include a term like

$$\begin{aligned} ill(c_V) &= iev(c_V)[1] \\ &= ((movi2s(c_V) \vee (movs2i(c_V)) \wedge (SA(c_V) \notin VSA) \vee \cdots \end{aligned}$$

The cause vector $ca(c_V, eev)$ is ANDed bit wise with a mask vector

$$mask(c_V)[j] = \begin{cases} SR[j] & \text{if } j \text{ maskable} \\ 1 & \text{otherwise} \end{cases}$$

in order to obtain the masked cause vector:

$$mca(c_V, eev)[j] = ca(c_V, eev) \wedge mask(c_V)[j]$$

The interrupt occurs if at least one masked cause bit is on

$$JISR(c_V, eev) = \bigvee_{j \in IV} mca(c_V, eev)[j]$$

Thus, although a virtual machine cannot read or write the status register SR, this register is nevertheless visible via the masked cause vector and the $JISR$-signal.

3 Physical machines

Physical machines are the sequential programming model of the hardware as seen by the programmer of an operating system kernel. Compared with the ISA of the virtual machine, more details are visible in configurations $c_P \in C_P$ of physical machines.

- All special purpose registers from Table 1 are visible. Formally $c_P.SPR : PSA \to \{0,1\}^{32}$ with $PSA = \{bin_5(a) : a \leq 13\}$ where $bin_n(a) \in \{0,1\}^n$ is the bitvector of length n such that $\langle bin_n(a) \rangle = a$. The mode register $c_P.SPR[10000] = c_P.mode$ distinguishes between system mode ($c_P.mode[0] = 1$) and user mode. In system mode reading and writing any special purpose register is legal.
- Page faults are visible, thus in the definition of the $JISR$-signal the full set of indices IP is used instead of IV.
- For physical machines the next state $\delta_P(c_P, eev)$ is defined in the case of an active signal $JISR(c_P, eev)$. See [MP00] for details. Similarly, in system mode physical machines can legally execute an `rfe` (return from exception) instruction.
- Instead of a uniform virtual memory the user now sees two memories: physical memory $c_P.pm$ and swap memory $c_P.sm$. The number of pages of available physical memory is specified by a parameter P.
- In user mode accesses to physical memory are translated.

In the remainder of this section we first specify a 1-level translation mechanism and the corresponding internal interrupt event signals $pff(c_P)$ and $pfls(c_P)$. Then we model I/O operations with the swap memory.

3.1 Address translation

In user mode, i.e. if $c_P.mode = 1$, virtual addresses $va = va.px \circ va.bx$ are transformed into three signals $pma(c_P, va), pff(c_P), pfls(c_P)$, where pff and $pfls$ are interrupt event signals and pma is the physical memory address in case the page fault signals stay inactive. Memory region $c_P.pm_{c_P.ptl \cdot 4+4}(c_P.pto \cdot 4K)$ is interpreted as the current page table. The page table entry address for virtual address va is defined as

$$ptea(c_P, va) = c_P.pto \cdot 4K + 4 \cdot va.px$$

and the corresponding page table entry is

$$pte(c_P, va) = c.pm_4(ptea(c_P, va)) \ .$$

These functions are only defined if there is no page table length exception

$$ptlexcp(c_P, va) = (\langle va.px \rangle > \langle c_P.ptl \rangle)$$

From the page table entry we extract the physical page index ppx, the valid bit v, and the protection bit p as suggested in Figure 1 by

$$\begin{aligned} ppx(c_P, va) &= pte(c_P, va)[31:12] \\ p(c_P, va) &= pte(c_P, va)[11] \\ v(c_P, va) &= pte(c_P, va)[10] \end{aligned}$$

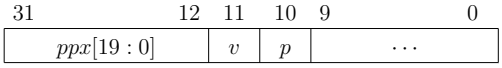

Figure 1: Page Table Entry

In case the valid bit is on, the physical memory address pma is obtained by concatenation of the physical page index with the byte index $va.bx$:

$$pma(c_P, va) = ppx(c_P, va) \circ va.bx$$

A page fault on fetch occurs, if page table length exception or invalid access occurs with virtual address dpc

$$pff(c_P) = ptlexcp(c_P, c_P.DPC) \vee \neg v(c_P, c_P.DPC)$$

In the absence of page faults on fetch we have in user mode now

$$I(c_P) = c_P.pm_4(pma(c_P, c_P.DPC))$$

In the absence of page faults on fetch a page fault on load store concerns virtual address $ea(c_P)$. Besides page table length exceptions and invalid access there might also be an attempt to perform a store operation, indicated by predicate $s(c_P)$, to a write protected page

$$pfls(c_P) = ptlexcp(c_P, ea(c_P)) \vee \neg v(c_P, ea(c_P)) \vee s(c_P) \wedge p(ea(c_P))$$

It is not difficult to specify multi-level translation can be formally specified in a similar way, see e.g. [Hil05, Chapter 5].

3.2 Modeling an I/O device

In order to handle page faults, one clearly has to be able to exchange pages between the physical memory $c_P.pm$ and the swap memory $c_P.sm$. For a (minimal) detailed treatment of this process one has to do four things:

1. Define I/O-ports as a portion of memory shared between the CPU and the I/O device holding the swap memory.
2. Specify the detailed protocol of the I/O-devices.
3. Construct a driver program say with the following three parameters passed on fixed addresses a_1, a_2, a_3 in physical memory: a physical page index parameter $ppxp(c_P) = c_P.pm_4(a_1)$, a swap memory page index parameter $spxp(c_P) = c_P.pm_4(a_2)$, and a physical-to-swap flag $p2s(c_P) = c_P.pm(a_3)$ indicating, whether a page is to be transferred from physical to swap memory ($p2s = 1$) or vice versa.
4. Show (among other things):[2] if the driver is started in configuration

[2]These are: (i) prove that program control returns (e.g. in case a jump and link instruction was used to call the driver $c'_P.DPC = c_P.GPR[11111], c'_P.PC = c_P.GPR[11111] + 4$), (ii) prove that except for appropriately defined book keeping information no other parts of the configuration changed, and (iii) the driver never leaves its own code region (needed to prove the correctness of the driver arguing only about the code of the driver).

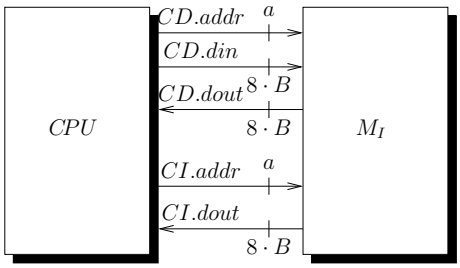

Figure 2: Old Memory Interface without MMUs

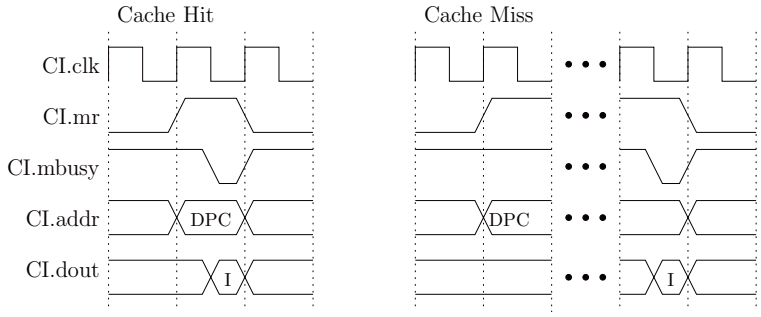

Figure 3: Timing Diagrams for Fetch Operations

c_P and never interrupted, then it eventually terminates in configuration c'_P with

$$c'_P.sm_{4K}(spxp(c_P) \cdot 4K) = c_P.pm_{4K}(ppxp(c_P) \cdot 4K) \text{ if } p2s(c_P) = 1$$
$$c'_P.pm_{4K}(ppxp(c_P) \cdot 4K) = c_P.sm_{4K}(spxp(c_P) \cdot 4K) \text{ if } p2s(c_P) = 0$$

Without this detailed treatment of I/O devices we have to assume the existence of a correct driver program as an axiom.

4 Construction and local correctness of MMUs

4.1 Notation

For cycles t and hardware signals or register contents x we denote by x^t the value of x during cycle t. We will refer to the hardware configuration by h. The components of this configuration are registers $h.R$ or memories $h.mem$. We often abbreviate $h.x$ by x.

4.2 Memory interface

We construct MMUs for processors with two first level caches, an instruction cache CI for fetches and a data cache CD for load / store instructions. Therefore the CPU communicates with the memory system via two sets of busses: one connecting the CPU with the instruction cache and the other one with the data cache (see Figure 2). We assume that the same protocol is used on both busses. Examples of the protocol are shown in Figure 3 for an instruction fetch with and without a cache hit. The essential properties of the bus protocol and the memory system are the following:

1. Accesses last from the activation of a read or write request signal (in the example mr) until one cycle after the busy signal is turned off; if the busy signal is not turned on at all, accesses last a single cycle.
2. Read and write requests are not allowed to be given simultaneously.
3. For the duration of an access, inputs from the CPU to the memory system must be held constant.
4. Liveness: if Conditions 2 and 3 are fulfilled, every access eventually ends.
5. Shared memory semantics: for cycles t and addresses a define $last(a,t)$ as the last cycle t' before t, when a write access to address a (necessarily on the data cache via bus $DC.din$) ended. Now assume a read access to cache $X \in \{CI, CD\}$ with address a ends in cycle t. Then the result on bus $X.dout$ is

$$X.dout^t = DC.din^{last(a,t)}$$

The definition permits to define the state of the two port memory system memory system $m(h)$ at time t by

$$m(h)^t(a) = DC.din^t last(a,t)$$

For a formal and complete version of this definition[3] see pages .... of [Bey04]. For a construction of a split cache system and a transcript of a formal proof, that it satisfies this specification, see [Bey04], pages 1–110. Guaranteeing that the CPU keeps inputs constant (Condition 3) during *all* accesses requires the construction of so-called *stabilizer circuits*, both for the instruction port and for the data port of the memory system. These circuits are needed for instance, if a fetch is in progress and simultaneously special addresses are forced into the PC and DPC due to an interrupt recognized deeper down in the pipeline. In this case the started access is completed with latched versions of the PC and DPC. If this precaution is not taken, liveness (Condition 4) is not guaranteed. For details see Chapter 4.4 of [Bey04].

[3] $last$ does not always exist

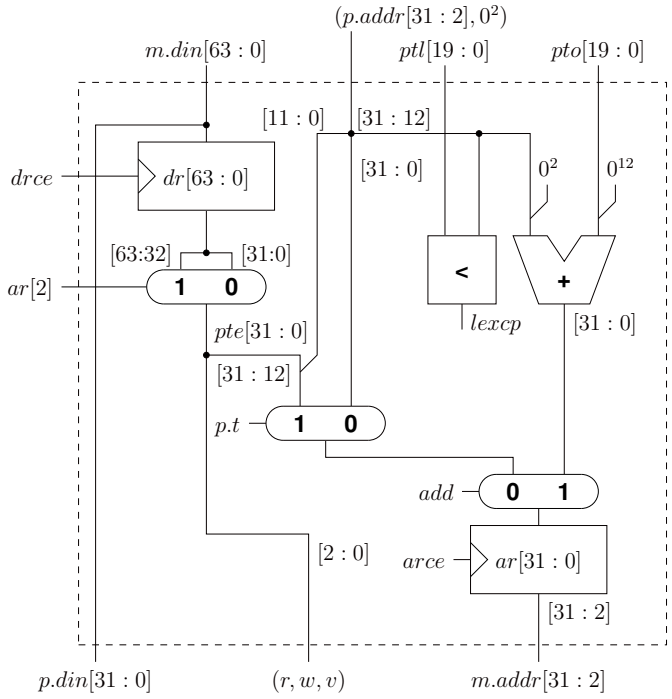

Figure 4: MMU Datapaths.

4.3 MMU construction and operating conditions

Figures 4 and 5 show datapaths and control automaton of a straightforward non optimized construction of an MMU. Two copies of this MMU are placed between the CPU and the two caches as shown in Figure 6. Note that the interface to the memory system is eight bytes wide and the address width is only 29 bits.

In system mode this MMU will only perform address translation under non trivial operating conditions. Consider an access of the CPU to the MMU lasting from a start cycle ts to an end cycle $te > ts$. We have to require that no signal or register content x from the four groups listed below changes its value during the access, so for all $\in \{ts, \ldots, te\}$ we have $x^t = x^{ts}$.

G1. Inputs from the CPU to the interface busses of the MMU; these are $p.dout$, $p.addr$, $p.mr$, and $p.mw$.

G2. The CPU registers $h.mode$, $h.pto$, and $h.ptl$ relevant for translation.

G3. In case of a translated accessing the page table entry used for translation, the shared memory content $m(h)_4(ptea)$ with $ptea = h.pto \cdot 4096 + 4 \cdot p.addr^{ts}.px$.

G4. In case of read access with physical address pa, the shared memory content $m(h)_8(pa)$.

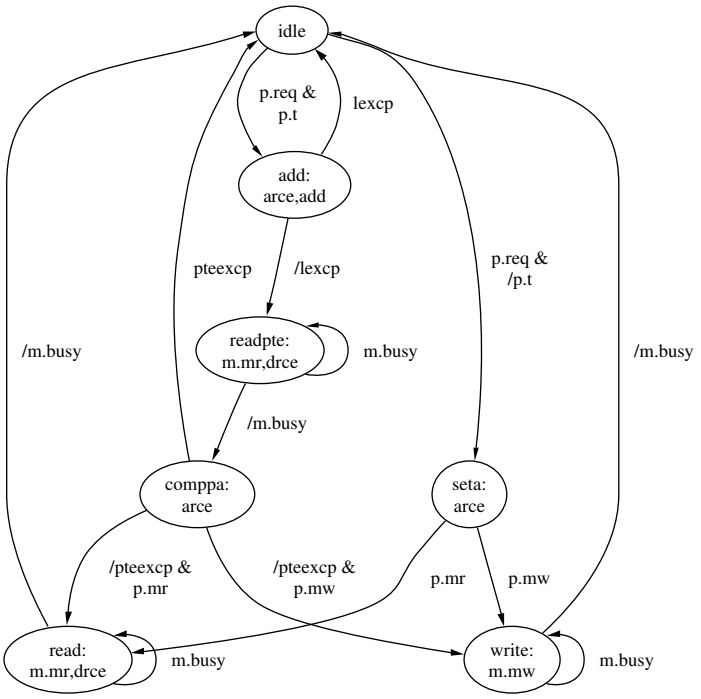

Figure 5: MMU Control Automaton. The signal $p.req = p.mw \vee p.mr$ indicates an ongoing request.

Using definitions analogous to Section 3.1 one can define for hardware configurations h and virtual addresses va a page table entry address $ptea(h, va)$ a page table entry $pte(h, va)$ and a physical memory address $pma(h, va)$. Note that under the operating conditions the virtual address va, the translation $pma(h, va)$ and in case of a read the data read from the shared memory stay the same during the whole access.

Assuming these operating conditions, we outline a fairly straightforward correctness proof for the MMU in the next subsection. Guaranteeing the operating conditions will be a considerably tougher issue.

4.4 Local MMU correctness

There is an obvious case split on the kind of access (i) read / write (ii) translated / untranslated (iii) with / without exception. We treat here only the case of a translated read access without exception.

First we identify the sequence of states in the control automaton for such a request. For states s we denote by s^+ the fact that control stays in state s until the busy signal is taken away from the memory interface.

Lemma 1 (Path lemma) *In a translated read without exception the path*

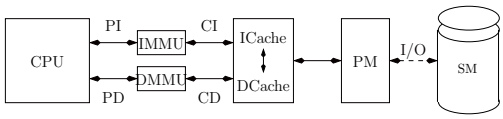

Figure 6: Processor and MMUs

through the control automaton is

$$idle \to add \to readpte^{+} \to comppa \to read^{+} \to idle \ .$$

Lemma 2 (Local correctness) *The result $p.din^{te}$ of a translated read without exception to a virtual address supplied as $va = p.addr^{ts}$ is*

$$p.din^{te} = m(h^{ts})_8(pma(h^{ts}, va \circ 0^3))) \ .$$

Proof. In cycle $ts + 1$ control is in state *add*. Hence in cycle $ts + 2$ we have in the MMU's address register ar the address of the required page table entry

$$\begin{aligned} ar^{ts+2} &= h.pto^{ts+1} + 4 \cdot p.ad^{ts+1}.px \\ &= h.pto^{ts} + 4 \cdot p.ad^{ts}.px \end{aligned}$$

by $G1$ and $G2$. Consider the cycle $t' \geq ts + 3$ when control is in state *readpte* and the busy signal from the memory interface is zero. Using $G3$ one argues that the MMU's data register dr contains in cycle $t' + 1$ the page table entry needed for the translation of va. If $ptea(h^{ts}, va)[2] = 1$, it is in the upper half of the data register. Otherwise, it is in the lower half.

$$pte(h^{ts}, va) = \begin{cases} dr^{t'+1}[63:32] & \text{if } ptea(h^{ts}, va)[2] = 1 \ ; \\ dr^{t'+1}[31:0] & \text{otherwise.} \end{cases}$$

One cycle later the MMU's address register ar contains the translated address

$$ar^{t'+2} = pma(h^{ts}, va)$$

Consider the cycle $t'' \geq t' + 2$ when control is in state *read* and the busy signal from the memory interface is zero. Using $G4$ one argues that the memory output *mdout* contains in cycle $te = t'' + 1$ the required result of the translated read access

$$p.dout^{te} = m.dout^{te} = m(h^{ts})_8(pma(h^{ts}, va \circ 0^3)))$$

5 Guaranteeing the operating conditions

Stable inputs from the CPU to the MMU's (Condition $G1$) can be guaranteed by using the stabilizer circuits mentioned in Section 4.2 with very modest enhancements. Condition $G4$ for loads can be guaranteed, if loads and stores are performed in order by the memory unit. Guaranteeing the remaining operating conditions (Conditions $G2$, $G3$, and $G4$ for fetch) requires a software convention *and* a hardware construction.

5.1 Software Synchronization Convention

We consider sequential computations of the physical machine $(c_P^0, c_P^1, \ldots)$; formally this means that we have for all steps i:

$$c_P^i = \delta_P(c_P^{i-1}, eev^i)$$

Recall that for physical machines the address $iaddr(c_P)$ from which an instruction is fetched depends on $c_P.mode$:

$$iaddr(c_P) = \begin{cases} c_P.DPC & \text{if } c_P.mode = 0\ ; \\ pma(c_P, c_P.DPC) & \text{otherwise.} \end{cases}$$

The instruction $I(c_P)$ executed in configuration c_P is then

$$I(c_P) = c_P.pm_4(iaddr(c_P))$$

We define an instruction as *synchronizing* if it is completed and the pipeline of the processor is drained before the (translation of the) fetch of the sequentially next instruction starts. The VAMP processor has already a synchronizing instruction, namely a `movs2i` instruction with $IEEEf$ as a source register.[4] We now also define the `rfe` instruction to be synchronizing. With the help of function $I(c_P)$ one defines a predicate $syncing(c_P)$ stating that the instruction executed in configuration c_P is synchronizing. The software sync-convention now has two parts:

1. Let $t < t'$. Assume $I(c_P^t)$ writes to $iaddr(c_P^{t'})$. Then for some t'' between t and t' instruction $I(c_P^{t''})$ must be synchronizing, i.e. we have $syncing(c_P^{t''})$. The corresponding condition is also needed without address translation in order to prevent the modification of an instruction in pipelined machines after it has been (pre-) fetched [SH98, BJK$^+$03].
2. Let $t < t'$. Assume instruction $I(c_P^{t'})$ is in user mode $(c_P^{t'}.mode = 1)$ and instruction $I(c_P^t)$ writes a page table entry read during the fetch of $I(c_P^{t'})$; the address of this entry is $ptea(c_P^t.DPC)$ for fetch. Then we also require $syncing(c_P^{t''})$ for an instruction t'' between t and t'.

Clearly, the first sync-convention addresses operating conditions $G4$ in case of a fetch, whereas sync Condition 2 addresses $G3$. In the hardware one has to address operating condition $G2$ and one has to implement the flushing of the processor once a synchronizing instruction is decoded.

5.2 Hardware mechanisms for synchronization

The VAMP processor has a two stage pipeline for instruction fetch and instruction decode, followed by a Tomasulo scheduler. For details see [BJK$^+$03, Kro01, Bey04]. Thus, there are many register stages S, e.g. IF for instruction fetch, PCs with PC and DPC, ID for instruction decode, $RS(rs)$ for reservation station rs, $ROB(tag)$ for the content of the reorder

[4] This instruction reads out the floating point interrupts accumulated *so far*.

buffer with address tag, RF for the register files, etc. In particular the instruction register I belongs to stage ID.

The clocking and stalling of individual stages is achieved by a so-called *stall engine*. For an introduction to stall engines see [MP00]; for better stall engines see [Kro01, Bey04]. We enhance here the stall engine from [Bey04].

Three crucial data structures resp. signals are associated with each stage S in the stalling engine:

1. The full bit $full_S$. It is on, if stage S has meaningful data. Clearing the bit $full_S$ flushes the stage. Here we will only be concerned with bit $full_{ID}$ of the instruction decode stage.
2. The local busy signal $busy_S$. If this signal is on in cycle t, then the circuits with inputs from register stage S do not produce meaningful data at the end of cycle t. Here we will only be concerned with the busy signal $busy_{IF}$ of the instruction fetch stage.
3. The update enable signals ue_S. It is like a clock enable signal. If ue_S is active in cycle t, the stage S has new data in cycle $t+1$. We will use these signals in Section 6.1 for the definition of scheduling functions.

Let $busy'_{IF}$ be the busy signal of the instruction fetch stage of a machine without memory management units. We define a new busy signal by

$$busy_{IF}(h) = busy'_{IF}(h) \vee \neg fetch(h)$$

where signal $fetch(h)$ is almost the read signal for the instruction MMU of the CPU.[5]

Signal $fetch$ is turned on, if (i) no instruction changing registers *pto*, *ptl* and *mode* is in progress and (ii) no synchronizing instruction is in progress. Instructions in progress can be in the instruction decode stage, i.e. in the instruction register IR, or they are issued but not completed, thus they are in the Tomasulo scheduler and its data structures. In a Tomasulo scheduler an instruction in progress which changes a register r from a register file is easily recognized by an inactive valid bit $r.v$. Thus we define

$$fetch(h) = h.pto.v \wedge h.ptl.v \wedge h.mode.v \wedge fetch'(h)$$

where function $fetch'(h)$ has to take care of instructions in the decode stage. Using predicates like $rfe()$ which are already defined for configurations also for the contents of the instruction register, we set

$$fetch'(h) = \neg(h.full_{ID} \wedge (syncing(IR) \vee movi2s(IR))) \ .$$

In the VAMP processor synchronizing instructions stay in the instruction decode stage until the can immediately proceed to the write-back stage.

[5] It is latched for the completion of an interrupted access

6 Processor correctness

6.1 Correctness criteria

We are using correctness criteria based on scheduling functions from [MP00, Kro01, Bey04, SH98]. Register stages S of the hardware come in three flavours:

- Visible stages (with respect to the physical machine): these stages are (i) PCs with the program counters $h.PC, h.DPC$, (ii) RF with the register files and $h.GPR$, $h.SPR$, and $h.FPR$ (iii) stage mem' with the user visible memory. This latter stage does not exist directly as a hardware component; instead it is simulated by the memory system (main memory and caches) and its state in hardware configuration h is encoded in function $m(h)$ (cf. Section 4.2).
- Invisible stages: the registers of these stages store values of auxiliary functions used in the definition of the sequential semantics of the physical machines. E.g. stage ID with the instruction register stores values $I(c_P)$, stage mem with the input registers to the memory system for load / store operations stores $ea(c_P)$, and so on.
- Stages from the data structures of the Tomasulo scheduler.

We map hardware stages S and hardware cycles t to instruction numbers i by means of scheduling functions: $sI(S,t) = i$. The intention is to relate configurations h^t of the hardware with configurations c_P^i of the specification machines in the following way:

1. For visible registers R from stages $S \neq mem'$:
$$h^t.R = c_P^{sI(t,S)}.R$$
Thus the specified value of visible hardware register R is the same as the value of R in the specification machine before execution of the i'th instruction, where $i = sI(S,t)$ is the instruction scheduled in stage S during cycle t.
The next two condition differ purely by notation:
2. For stage mem':
$$m(h^t) = c_P^{sI(mem',t)}.pm$$
3. For invisible registers R in stage S:
$$h^t.R = R(c_P^{sI(S,T)})$$
4. Specific correctness criteria are used for the data structures of the Tomasulo scheduler. For details see [Kro01].

The three main definitions for scheduling functions which make this work are the following:

1. In order fetch: The fetch scheduling function is incremented if instruction decode stage receives a new instruction. Recall that ue_{ID} is the update enable function of the instruction register:
$$sI(fetch, t+1) = \begin{cases} sI(fetch,t) + 1 & \text{if } ue_{ID}^t \ ; \\ sI(fetch,t) & \text{otherwise.} \end{cases}$$

2. The scheduling of a stage S' that is not updated does not change:

$$ue^t_{S'} = 0 \rightarrow sI(S', t+1) = sI(S', t)$$

3. If data are clocked in cycle t from stage S' to S (a formal definition depends on update enable signals, mux select signals, or driver enable signals during cycle t) then

$$sI(S', t+1) = \begin{cases} sI(S,t) + 1 & \text{if } S' \text{ invisible;} \\ sI(fetch, t) & \text{if } S' \text{ visible.} \end{cases}$$

Thus intuitively an instruction number $i = sI(S,t)$ moves together with the data through the datapath; upon reaching a register in a visible stage S' however, the register receives the value *after* the i-th instruction, i.e. before instruction $(i+1)$-th instruction.

6.2 Correctness proof with external interrupt signals

In general the hardware of a pipelined processor does not complete one instruction per cycle. As there are more cycles t than instructions i there are necessarily more external interrupt events signals eev^t_h 'seen' by the hardware than event signals eev^i seen by the sequential specification machine. As the computation of the sequential machine is defined by

$$c^{i+1}_P = \delta_P(c^i_P, eev^i)$$

one has to define the interrupt signal eev^i seen by the specification machine from the signals eev^t_h seen by the hardware and the processor. This has already been observed in [SH98, MP00].

The VAMP processor samples external interrupt signals only during the write-back stage WB (i.e. it does not sample them all). Every instruction i is only for one cycle t in the write-back stage. Call this cycle $t = WB(i)$. The correctness proof then works with

$$eev^i = eev^{WB(i)}_h \ .$$

That no harm is done by the external interrupts signals, which are ignored both by the sampling of the hardware and the sequential programming model, is a problem that has to be solved by the protocol between the processor and the I/O devices.

6.3 Correctness proof

We give the new part of the processor correctness proof for a translated instruction fetch without exceptions. The other new cases are handled by similar arguments. Thus consider a translated read access on the instruction port of the CPU which lasts from cycle ts to cycle te, let $i = sI(fetch, ts)$ and let $t \in \{ts, \ldots, te\}$ be any cycle of the access. From the correctness proof for the processor hardware we conclude, that on the

address bus of the instruction MMU $PI.addr$ we have observed during cycle t the correct delayed PC,

$$PI.addr(h^t) = c_P^i.DPC[31:2] \ .$$

Let $i'(t) = sI(RF, t) < i$ be the instruction in the register stage during cycle t. Also by processor correctness we have for all registers R of the register files

$$h^t.R = c_P^{i'(t)}.R$$

In particular this holds for registers $R \in \{pto, ptl, mode\}$. By the construction of the fetch signal all instructions $x < i$ that update special purpose register pto, ptl, or $mode$ have already left the pipe already at cycle ts (and, because we fetch in order no instructions $x > i$ can enter the pipe while instruction i is in the fetch stage). We conclude for registers $R \in \{pto, ptl, mode\}$ and all cycles $t \in \{te, \ldots, ts\}$ that

$$c_P^i.R = c_P^{i'(t)}.R = h^t.R \ .$$

Let $i''(t) = sI(mem', t)$. From processor correctness we get

$$m(h^t) = c_P^{i''(t)}.pm$$

By the second part of the software sync-convention all instructions $x < i$ which write the page table entry address $ptea(c_P^i, c_P^i.DPC)$ have left the pipe already at cycle ts. We conclude

$$\begin{aligned} pte(c_P^i, c_P^i.DPC) &= c_P^i.pm_4(ptea(c_P^i, c_P^i.DPC)) \\ &= c_P^{i''(t)}.pm_4(ptea(c_P^i, c_P^i.DPC)) \\ &= m(h^t)_4(ptea(c_P^i, c_P^i.DPC)) \end{aligned}$$

By the first part of the software sync-convention all instructions which write the physical memory address $pma(c_P^i, c_P^i.DPC)$ have also left the pipe already at cycle ts. As above we conclude

$$\begin{aligned} I(c_P^i) &= c_P^i.pm_4(pma(c_P^i, c_P^i.DPC)) \\ &= c_P^{i''(t)}.pm_4(pma(c_P^i, c_P^i.DPC)) \\ &= m(h^t)_4(pma(c_P^i, c_P^i.DPC)) \end{aligned}$$

Hence the operating conditions for the MMU are fulfilled, and we get from the local correctness lemma:

$$PI.dout(h^{te}) = m(h^{ts})_8(pma(h^{ts}, PI.addr(h^{ts}) \circ 0^3))$$

The rest is lengthy but trivial. Specializing the equations above for $t = ts$ gives

$$\begin{aligned} PI.addr(h^{ts}) &= c_P^i.DPC[31:2] \\ h^{ts}.pto &= c_P^i.pto \\ pte(c_P^i, c_P^i.DPC) &= m(h^{ts})_4(ptea(c_P^i, c_P^i.DPC)) \\ I(c_P^i) &= m(h^{ts})_4(pma(c_P^i, c_P^i.DPC)) \end{aligned}$$

Using the definition of functions $ptea$, pte and pma for hardware machines we conclude successively

$$\begin{aligned}
ptea(h^{ts}, c_P^i.DPC) &= h^{ts}.pto + 4 \cdot c_P^i.DPC.px \\
&= c_P^i.pto + 4 \cdot c_P^i.DPC.px \\
&= ptea(c_P^i, c_P^i.DPC) \\
pte(h^{ts}, c_P^i.DPC) &= m(h^{ts})_4(ptea(h^{ts}, c_P^i.DPC)) \\
&= m(h^{ts})_4(ptea(c_P^i, c_P^i.DPC)) \\
&= pte(c_P^i, c_P^i.DPC) \\
pma(h^{ts}, c_P^i.DPC) &= pte(h^{ts}, c_P^i.DPC).px \circ c_P^i.DPC.bx \\
&= pte(c_P^i, c_P^i.DPC).px \circ c_P^i.DPC.bx \\
&= pma(c_P^i, c_P^i.DPC) \\
PI.dout(h^{te}) &= m(h^{ts})_8(pma(h^{ts}, PI.addr(h^{ts}) \circ 0^3)) \\
&= m(h^{ts})_8(pma(h^{ts}, c_P^i.DPC[31:2] \circ 0^3)) \\
&= m(h^{ts})_8(pma(c^i, c_P^i.DPC[31:2] \circ 0^3)) \\
&= c_P^i.pm_8(pma(c^i, c_P^i.DPC[31:2] \circ 0^3)) \\
I(c_P^i) &= c_P^i.pm_4(pma(c^i, c_P^i.DPC)) \\
&= \begin{cases} PI.dout(h^{ts})[63:32] & \text{if } c_P^i.DPC[2]\ ; \\ PI.dout(h^{ts})[31:0] & \text{otherwise.} \end{cases}
\end{aligned}$$

Thus, by selecting the right half in the double word $PI.dout(h^{te})$ using bit 2 of the delayed program counter, in cycle $te+1$ we clock $I(c_P^i)$ in the instruction register I. We have $sI(ID, te+1) = i$. Thus for a translated fetch without exceptions lasting from cycle ts to cycle te we have shown the required hardware correctness statement

Lemma 3 $h^{te+1}.I = I(c_P^i) = I(c_P^{sI(ID,te+1)})$

7 Virtual machine simulation

In this section we outline an *informal* proof that a physical machine with an appropriate page fault handler can simulate a virtual machine. We will use pseudo code as well as C like data structures in order to describe the handler and we will argue in the style of an efficient algorithms paper. Making these arguments precise is not trivial; we give some details in the section on further work.

For page indices px we define the physical page px, the swap page px, and the virtual page px respectively as

$$\begin{aligned}
ppage(c_P, px) &= c_P.pm_{4K}(px \circ 0^{12})\ , \\
spage(c_P, px) &= c_P.sm_{4K}(px \circ 0^{12})\ , \text{ and} \\
vpage(c_V, px) &= c_V.vm_{4K}(px \circ 0^{12})\ .
\end{aligned}$$

We extend the definition of physical page indices $ppx(c_P, va)$ and valid bits $v(c_P, va)$ from addresses va to page indices px by

$$\begin{aligned}
ppx(c_P, px) &= ppx(c_P, px \circ 0^{12}) \text{ and} \\
v(c_P, px) &= v(c_P, px \circ 0^{12})\ .
\end{aligned}$$

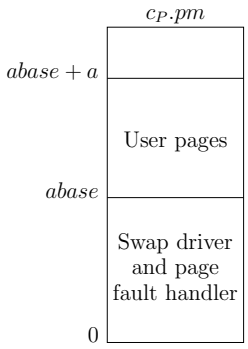

Figure 7: Memory Map with addresses given as page indices.

7.1 Memory map of the physical machine

We partition the physical memory $c_P.pm$ into user memory and system memory according to Figure 7. Starting at the physical address with page index $abase$ we allocate a pages of user memory. This defines the set of user page indices

$$UP = \{u \mid abase \leq \langle u \rangle < abase + a - 1\} \ .$$

Physical addresses with page indices smaller than $abase$ are used by the page fault handler and the swap memory driver.

We list below the data structures used by the handler and some invariants for them.

- A process control block PCB to save the processor registers of the virtual processor when the processors runs in system mode.
- The page table

  $$PT(c_P) = c_P.pm_{4 \cdot V}(c_P.pto \cdot 4K)$$

  where $V = \langle c_P.ptl \rangle + 1$ is the number of accessible virtual pages.

  We require for all virtual page indices $0 \leq px < V$ that the corresponding physical page index belongs to a user page if it is valid,

  $$v(c_P, px) \implies ppx(c_P, px) \in UP \ .$$

- the physical page index MRL of the most recently loaded page.
- A swap page index $sbase$ in swap memory. For virtual addresses va we will use the swap memory address

  $$sma(c_P, va) = sbase \cdot 4K + va$$

- A user page index $b \in \{0, \ldots, a-1\}$. We call a user page $u \in \{0, \ldots, a-1\}$ full if it stores a valid virtual page, i.e. if there is a virtual page index $0 \leq px < V$ such that

  $$v(c_P, px) \wedge ppx(c_P, px) = abase + u$$

A user page which is not full is called free. We maintain as an invariant that user page u is free iff $u > b$.

- an array B of size a holding virtual page indices. If $u \leq b$, i.e. if user page u is full, then

$$abase + u = ppx(c_P, B[u])$$

This data structure is used for victim selection, if a page has to be evicted.

- Parameters $ppxp$, $spxp$, and $p2s$ of the driver for the swap memory.

7.2 Simulation relation

For virtual machine configurations c_V and physical machine configurations c_P we define a simulation relation $B(c_P, c_V)$ stating that configuration c_P is an encoding for configuration c_V. We require that the invariants of the previous subsections hold for the physical machine and that the physical machine is in user mode ($c_V.mode = 1$). The simulation relation is composed of two parts:

1. For all virtual addresses va

$$c_V.p(va) = p(c_P, va) \ .$$

i.e. the write protection function is encoded in the protection bits of the page tables.

2. For all virtual addresses va

$$c_V.vm(va) = \begin{cases} c_P.pm(pma(c_P, va)) & \text{if } v(c_P, va) \ ; \\ c_P.sm(sma(c_P, va)) & \text{otherwise.} \end{cases}$$

So, the user pages of physical memory act as a (write-back) cache for the swap memory. This condition may be equivalent formulated for all virtual page indices px as

$$vpage(c_V, px) = \begin{cases} ppage(c_P, ppx(c_P, px)) & \text{if } v(c_P, px) \ ; \\ spage(c_P, sbase + px) & \text{otherwise.} \end{cases}$$

7.3 Page fault handler and software conditions

Assume the interrupt occurred during physical machine configuration c_P encoding virtual machine configuration c_V, i.e. we have $B(c_P, c_V)$.

We describe a very simple handler; the handler itself is never interrupted. Thus it suffices that the handler saves only the general purpose registers of the physical processor into the process control block. Testing the exception cause register ECA it is easy to determine, whether a page fault on fetch or a page fault on load store has occurred. For the former we have $ECA[3] = 1$, for the latter we have $ECA[4] = 1$.

It is easy to specify and implement the handler in case of an exception due to page table length exception or rights violations: the exception

is recognized and the simulation is stopped. Thus assume a page fault occurs, because the required virtual page is not in physical memory. The virtual address xva causing the exception is defined as

$$xva = \begin{cases} EDPC & \text{if } pff \ ; \\ EDATA & \text{if } pfls \ . \end{cases}$$

We call the page index of the exception virtual address the exception virtual page $xv = xva.px$. Because this page is not in memory and $B(c_P, c_V)$ holds, we know that the correct virtual page is stored in swap memory

$$spage(c_P, sbase + xv) = vpage(c_V, xv)$$

If $b = a-1$ there are no free user pages and a victim physical page index vp is selected from the user pages. However, the most recently loaded page is never chosen as victim to avoid deadlock, so $vp \in UP \setminus \{MRL\}$.

Let $vp = abase + u$. By looking up table entry $B[u]$ we determine the corresponding victim virtual page index $vv = B[u]$ of the virtual page stored at physical page vp.

Because $B(c_P, c_V)$ holds and $ppx(c_P, vv) = abase + u$ we have

$$\begin{aligned} vpage(c_V, vv) &= ppage(c_P, ppx(c_P, vv)) \\ &= ppage(c_P, ppx(c_P, B[u])) \\ &= ppage(c_P, abase + u) \\ &= ppage(c_P, vp) \end{aligned}$$

Running the driver with parameters

$$(ppxp, spxp, p2s) = (vp, sbase + vv, 1)$$

will save the victim page to swap memory, i.e. we end up in a physical machine configuration $c_P^{(1)}$ with

$$spage(c_P^{(1)}, sbase + vv) = ppage(c_P, vp) \ .$$

We conclude

$$spage(c_P^{(1)}, sbase + vv) = ppage(c_P, vp) = vpage(c_V, vv) \ .$$

Hence, if we clear the valid bit of page vv and enter a configuration $c_P^{(2)}$ with $v(c_P^{(2)}, vv) = 0$, the simulation relation $B(c_P^{(2)}, c_v)$ still holds.

We now have a user page index e where we can swap in the exception virtual page, namely

$$e = \begin{cases} abase + b + 1 & \text{if } b < a - 1 \ ; \\ vp & \text{otherwise.} \end{cases}$$

In the first case we increment b. Running the driver with parameters

$$(ppxp, spxp, p2s) = (e, sbase + xv, 0)$$

will swap the exception virtual page into physical memory, i.e. we end up in a configuration $c_P^{(3)}$ where

$$\begin{aligned} ppage(c_P^{(3)}, e) &= spage(c_P, sbase + xv) \\ &= vpage(c_V, xv) \ . \end{aligned}$$

Thus, $B(c_P^{(4)}, c_V)$ and the invariants will hold if we set

$$\begin{aligned} v(c_P^{(4)}, vx) &= 1 \\ ppx(c_P^{(4)}, vx) &= e \\ B[e - abase] &= vx \\ MRL &= e \end{aligned}$$

in a later configuration $c_P^{(4)}$. The handlers completes its work by restoring the general purpose registers from the process control block and executing an `rfe` instruction.

By inspection of the handler we see that the software synchronization conditions are fulfilled and thus we can conclude that with this handler the hardware specified above will work correctly.

7.4 Simulation theorem

Theorem 1 *For every computation $(c_V^0, c_V^1, \ldots)$ of the virtual machine there is a computation $(c_P^0, c_P^1, \ldots)$ of the physical machine and there are step numbers $(s(0), s(1), \ldots)$ such that we have for all t:*

$$B(c_V^t, c_P^{s(t)})$$

This means that t steps of the virtual machine are simulated after $s(t)$ steps by the physical machine. This is obviously shown by induction on t. For the initialization we set $b = -1$, i.e. all physical page are invalid, and the entire virtual memory is stored in swap memory. Concluding from t to $t + 1$ there are several cases. If there is no exception, then one step of the virtual machine is simulated by one step of the physical machine running in user mode[6] and we get

$$s(t + 1) = s(t) + 1$$

Because in a single instruction we can have up to two page faults, there remain four cases:

1. A single page fault on fetch.
2. A single page fault on load / store.
3. A page fault on fetch followed by a page fault on load / store.
4. A page fault on load / store followed by a page fault on fetch. This case occurs if the page with the current instruction is the victim page of the first page fault.

[6] Not in one cycle of the hardware!

For physical machine steps s let $\tau(s)$ be the first step s' after s such that the machine is in user mode:

$$\tau(s) = \min\{s' \mid s' > s \wedge c_P^{s'}.mode\}$$

In all four cases we have a page fault in step $s(t)+1$. In the first two cases the simulation of step $t+1$ succeeds without page fault in the first step in user mode after step $s(t)+1$ thus

$$s(t+1) = \tau(s(t)+1)+1 \ .$$

In the other two cases the second page fault occurs in step

$$s' = \tau(s(t)+1)+1 \ .$$

Because the victim page of the second page fault is not the page swapped in during the handling of the first page fault the simulation will succeed in step

$$s(t+1) = \tau(s')+1 \ .$$

8 Summary and further work

We have presented two main results. (i) We have reported about the formal verification of a processor with (simple) MMUs. The local correctness proof for an MMU alone (Section 4.4) is straight forward, but it hinges on nontrivial operating conditions. Guaranteeing the operating conditions requires a variety of arguments, from very detailed arguments about the hardware (e.g. Section 5.2) to the format of page fault handlers (Section 7.3). (ii) Arguing about low level system software we have given a paper and pencil proof for a simulation theorem stating that virtual machines are simulated by physical machines with appropriate page fault handlers. Because modern operating systems support multitasking and virtual memory, the results of this paper are crucial steps towards the verification of entire computers systems.

Presently we see three main directions for further work. (i) On the hardware side one wants to verify processors with pipelined MMUs, multi level page tables and table look aside buffers. (ii) On the low level software side one wants to formally prove the simulation theorem of this paper. This is part of an ongoing effort to formally verify an entire operating system kernel as part of the Verisoft project. Paper and pencil proofs can be found in [SysArch04] (iii) one wants to establish correctness theorems for the memory management mechanisms of shared memory multiprocessors. The thesis [Hil05] contains results of this nature.

References

[ACHK04] Mark Aagaard, Vlad Ciubotariu, Jason Higgins, and Farzad Khalvati. Combining equivalence verification and completion functions. In Alan Hu and Andrew Martin, editors, *FMCAD*, volume 3312 of *Lecture Notes in Computer Science*, pages 98–112. Springer, 2004.

[Bev89] William R. Bevier. Kit and the short stack. *Journal of Automated Reasoning*, 5(4):519–530, 1989. In [Boy89].

[Bey04] Sven Beyer. *Putting It All Together: Formal Verification of the VAMP*. PhD thesis, Saarland University, Computer Science Department, 2004.

[BJK+03] Sven Beyer, Christian Jacobi, Daniel Kroening, Dirk Leinenbach, and Wolfgang Paul. Instantiating uninterpreted functional units and memory system: functional verification of the VAMP processor. In Daniel Geist and Enrico Tronci, editors, *CHARME*, volume 2860 of *Lecture Notes in Computer Science*, pages 51–65. Springer, 2003.

[BM88] Robert S. Boyer and J S. Moore. *A Computational Logic Handbook*. Academic Press, 1988.

[Boy89] Robert S. Boyer, editor. *Special Issue on System Verification*, volume 5 of *Journal of Automated Reasoning*. Kluwer Academic Publishers, 1989.

[Hil05] Mark Hillebrand. *Address Spaces and Virtual Memory: Specification, Implementation, and Correctnesss (Draft)*. PhD thesis, Saarland University, Saarbrcken, Germany, 2005.

[Hun89] Warren A. Hunt. Microprocessor design verification. *Journal of Automated Reasoning*, 5(4):429–460, December 1989. In [Boy89].

[Kro01] Daniel Kroening. *Formal Verification of Pipelined Microprocessors*. PhD thesis, Saarland University, Computer Science Department, 2001.

[Moo89] J Strother Moore. A mechanically verified language implementation. *Journal of Automated Reasoning*, 5(4):461–492, 1989. In [Boy89].

[Moo03] J Strother Moore. A grand challenge proposal for formal methods: A verified stack. In Bernhard K. Aichernig and T. S. E. Maibaum, editors, *10th Anniversary Colloquium of UNU/IIST*, volume 2757 of *Lecture Notes in Computer Science*, pages 161–172. Springer, 2003.

[MP00] Silvia M. Mueller and Wolfgang J. Paul. *Computer Architecture: Complexity and Correctness*. Springer, 2000.

[OSR92] S. Owre, N. Shankar, and J. M. Rushby. PVS: A prototype verification system. In *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752. Springer, 1992.

[SH98] Jun Sawada and Warren A. Hunt, Jr. Processor verification with precise exeptions and speculative execution. In *CAV '98: Proceedings of the 10th International Conference on Computer Aided Verification*, pages 135–146. Springer-Verlag, 1998.

[Ver03] The Verisoft Project. `http://www.verisoft.de/`, 2003.

[You89] William D. Young. A mechanically verified code generator. *Journal of Automated Reasoning*, 5(4):493–518, 1989. In [Boy89].